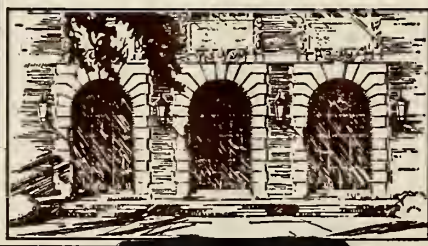


LIBRARY OF THE
UNIVERSITY OF ILLINOIS
AT URBANA-CHAMPAIGN

510.84
I l 6 r
no. 590-594
cop. 2





Digitized by the Internet Archive
in 2013

<http://archive.org/details/designofalgorithm592dono>

math

*10.84
26N
D. 592
op. 2*

DESIGN OF
AN ALGORITHM FOR THE DISPLAY
OF VISIBLE SURFACES

by

Walt Donovan

October 1973



THE LIBRARY OF THE

JAN 9 1974

UNIVERSITY OF ILLINOIS
AT URBANA-CHAMPAIGN

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

UIUCDCS-R-73-592

DESIGN OF
AN ALGORITHM FOR THE DISPLAY
OF VISIBLE SURFACES

by

Walt Donovan

October 1973

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
URBANA, ILLINOIS 61801

* Supported in part by the Atomic Energy Commission under grant US AEC AT(11-1)2118 and submitted in partial fulfillment of the requirements of the Graduate College for the degree of Master of Science in Computer Science.

ACKNOWLEDGMENT

I am grateful to my advisor, Professor Bruce H. McCormick, for the opportunity to work on this problem, and also for the long hours of unstinting aid and advice he unhesitatingly gave me.

I appreciate the extra effort that Professor James N. Snyder went to to assure me of financial aid until this thesis had been completed.

The excellent typing was accomplished in a remarkably efficient manner by Mrs. June Wingler, who also managed to read my handwriting virtually errorlessly. Many thanks also to Stan Zundo who raced with time to draw most of the figures.

PREFACE

A visible surface algorithm capable of showing on a raster display smooth, transparent objects without jagged edges is described. The polygons defining the object to be displayed are put into visible order, and then output to the raster frame buffer by a special face drawing and shading algorithm. Collections of objects can be manually put into visible order and each group drawn separately; this allows scenes with more faces than can be held in the program data structure at any one time. Minimum hardware for implementation is a raster display with a frame buffer, and a minicomputer with a disk.

TABLE OF CONTENTS

	Page
1. INTRODUCTION.....	1
2. PREVIOUS WORK.....	7
3. VISIBLE SURFACE DETERMINATION.....	9
4. FACE SPLITTING.....	23
5. FACE SAMPLING.....	32
6. SEGMENT SHADING.....	39
7. CONCLUSIONS.....	51
APPENDIX.....	52
LIST OF REFERENCES.....	55

1. INTRODUCTION

The problem we are concerned with is that of producing, by computer, realistic shaded renderings of three-dimensional objects. Such a facility is of great use for representing mathematical functions, in computer-aided design, producing architectural renditions, for computer-generated movies and animation, and in aircraft or traffic simulation.

Many people have worked on this problem [1-8], and excellent pictures have resulted. All of these algorithms use the basic idea of determining shading only at regularly-spaced sample points, and using the resulting values to approximate the correct view (see Figure 1). That is, the surface that is visible at each sampling point is found, and the appropriate shading value is determined therefrom. The display output of these algorithms, though, suffer from sampling error, which causes the straight edge of Figure 2a to appear as in 2b, and object A of Figure 2c to disappear, even though object B is considered visible.

Our approach to the problem has resulted in an algorithm that can display smooth, transparent objects with no detectable sampling error.

The objects we use are either comprised of or approximated by planar polygons; each polygon has photometric

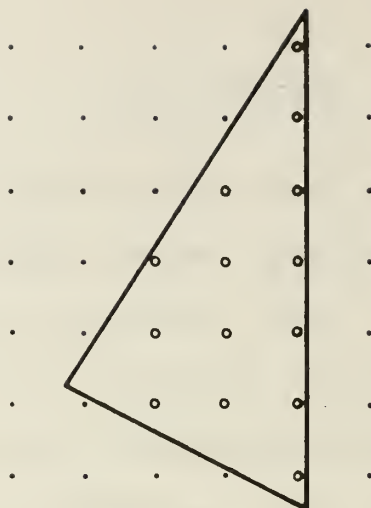


Figure 1. Sampled triangle: circles inside triangle denote sampling points used to represent triangle..

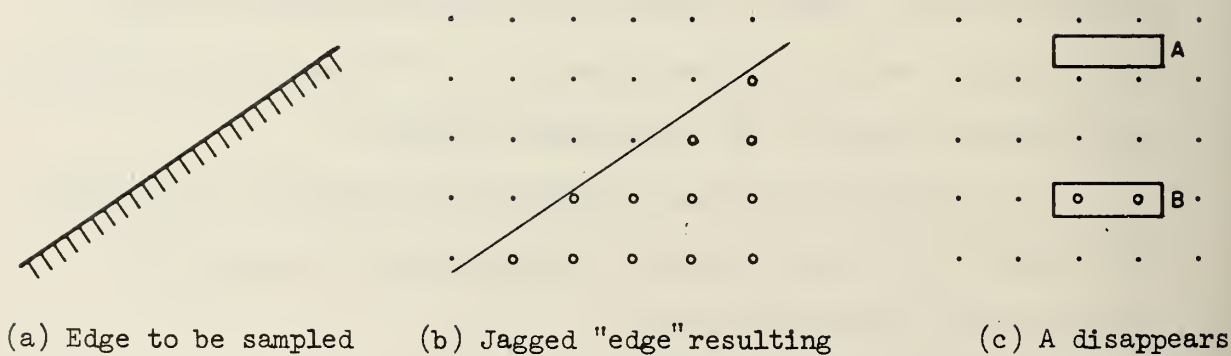
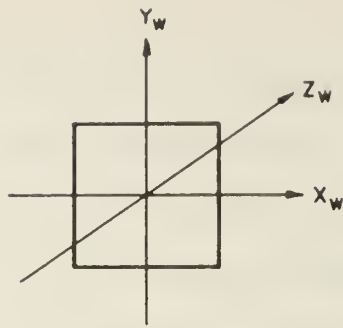


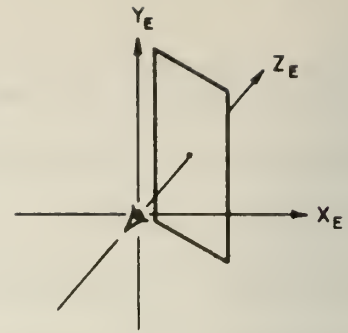
Figure 2. Sampling Error

data associated (transmittivity, reflectivity) and whether or not it is part of a smooth surface. The polygons are defined in world coordinates (X_w, Y_w, Z_w) (Figure 3a); they are then transformed to the eye coordinate system (X_e, Y_e, Z_e) with the eye at the origin and looking down the $-Z$ axis (Figure 3b). Polygon parts outside the pyramid of vision (Figure 3c) are removed and the remaining portions transformed to the projective or screen coordinate system (X_s, Y_s, Z_s) (Figure 3d). All visibility comparisons are done in the latter coordinates. The sampling grid is located in the $Z_s=0$ plane, and the mapping from screen coordinates to the sample plane is $(X_s, Y_s, Z_s) \rightarrow (X_s, Y_s, 0)$ (Figure 3e). Due to the clipping procedure above, X_s and Y_s for objects are in $[-1,1]^2$. The sample coordinate system $(X_{\text{samp}}, Y_{\text{samp}})$ is superimposed in the sample plane as in Figure 3f; an integer value for X_{samp} , say, means that the point lies on one of the x grid lines. Conversion from real to integer sample coordinates is handled in the shader. Sample points are sometimes just called points, picture points, elements, or "pixels"; a horizontal sample line ($Y_{\text{samp}}=n$) is called a scan line. Shading values for pixels are represented as non-negative integers measuring intensity; 0 means black or as dark as possible, and the maximum pixel value (restricted by the hardware) represents white or maximum brightness. Color can be represented by a 3-vector of shading values indicating the intensities of the red, blue, and green components. The pixel values obtained are stored



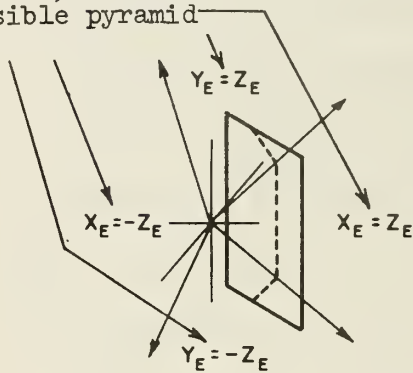
(a) World Coordinates
(unit square example)

Scaling,
rotation,
→
and
translation



(b) Eye coordinates, with eye at origin (translated in X and Z, rotated about Y axis)

Planes, defining visible pyramid

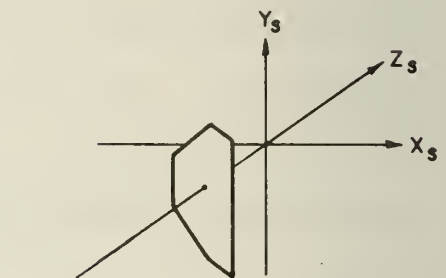


(c) Removing parts outside visible pyramid, to ensure that screen coordinates lie in $[-1,1]^2$ (cut on dashed line)

$$X_S = -\frac{X_E}{Z_E}$$

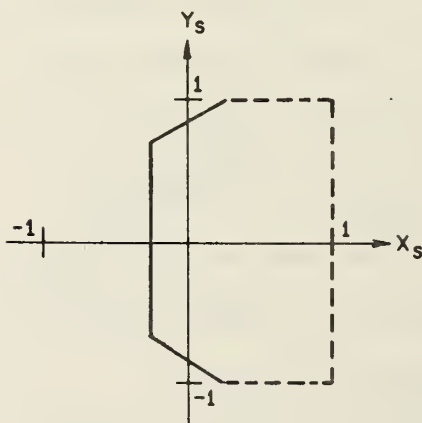
$$Y_S = -\frac{Y_E}{Z_E}$$

$$Z_S = -\frac{1}{Z_E}$$

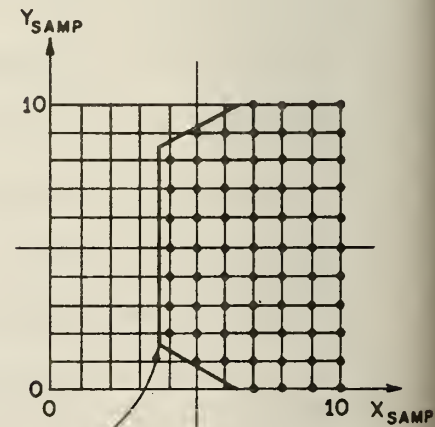


EYE AT
-∞

(d) Screen coordinates (projection of clipped square..lines map to lines and planes to planes)



(e) Sample Plane (orthogonal projection onto $Z_S = 0$ plane)



E (3.7, 1.8)

(f) Sampling grid and samples (note exact values of vertex E)

Figure 3. Coordinate Systems

in a raster frame buffer or just a large memory; this can then be displayed either on a television (raster) display or a precision CRT. The television raster lines correspond to the scan lines at the same position. Figure 4 is a sample shaded rendition.

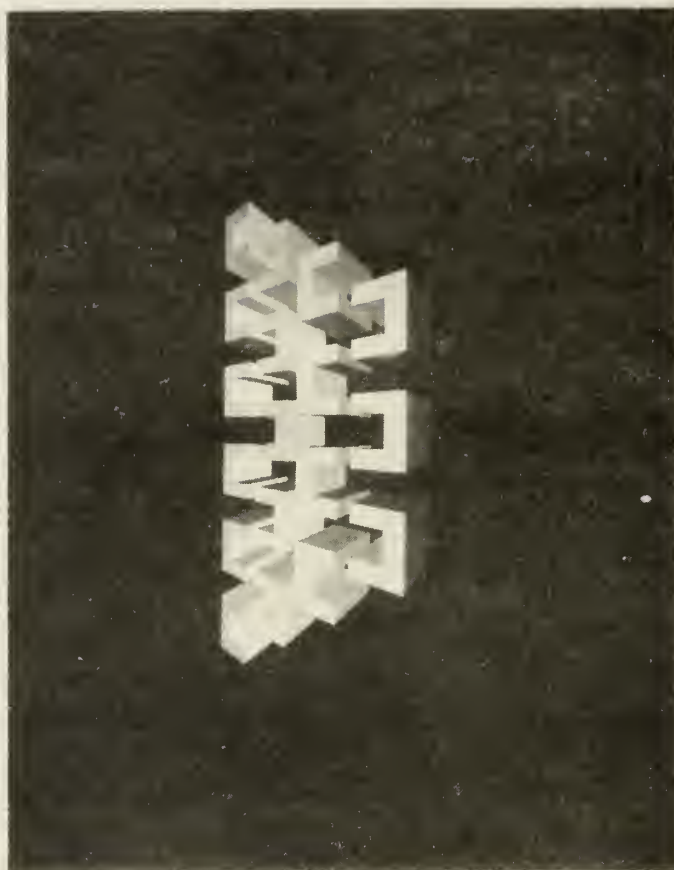


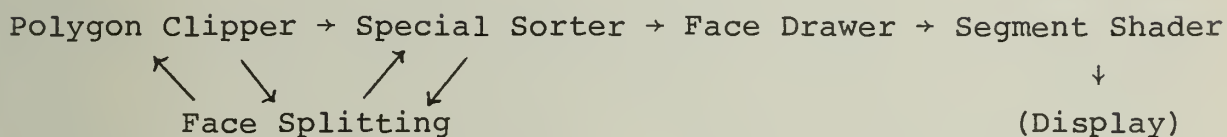
Figure 4. Example of a Shaded Rendition

2. PREVIOUS WORK

The display algorithm we have designed breaks into five parts, with the following flow of control:

(Start)

↓



The basic idea of ordering faces and using the frame buffer to delete hidden parts was first considered in a simplified manner suitable for hardware implementation by Schumacker [8]; Newell[1] later independently augmented the idea to a method that works in general. Our special sorter works in the same way as Newell's, but we have modified their non-deterministic strategy to a deterministic one to aid our justification and (informal) proof of it. The method we use to sample and draw faces is a slightly altered and corrected version of the one embedded in Watkin's visible surface algorithm [6]. The segment shader uses the same method to do smooth shading as found in Gourad [2], and to display transparent faces as found in Newell.

New methods described here are a shading algorithm that removes jagged edges and compensates for faces disappearing due to sampling error, and a face splitting algorithm

that can also be used for polygon clipping. Since they do not use any special property of the Newell special sorter, these algorithms may be easily adapted for use with implementations of Watkin's or any other similar visible surface display method.

In this thesis, we will start out first describing the special sorter so as to get an initial grasp on the visible surface problem. The face splitter required by the special sorter and the polygon clipper will then be discussed simultaneously, since they are closely connected. The face drawing routine is then described, both for completeness and also because it is affected by the shader. The special shader is the last algorithm to be covered because we must then consider the entire effect of the program up to that point for a proper understanding of its processing.

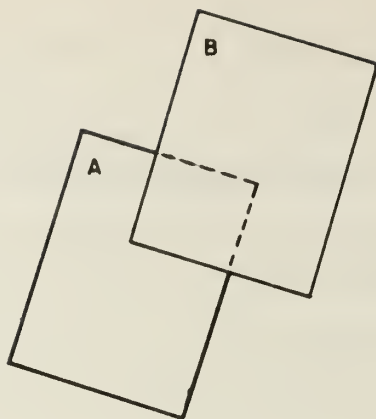
3. VISIBLE SURFACE DETERMINATION

The display problem can be divided into two steps: at each sample point, we must find out which face of those input is visible. Then, we must compute the correct shading value for the face at that sample point.

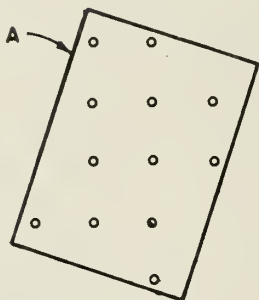
Our approach to visibility determination is suggested by the following experiment. Figure 5a shows the (line drawing) projection of two opaque squares, with the hidden part dashed. The picture can be considered to show a precedence relation between squares A and B, where $A < B$ because B occults part of A (and A does not occult any part of B). The significance of this precedence is shown in Figures 5b-c. Since $B > A$, we sample and shade (draw, henceforth) A and put the result into the frame buffer first. Then we draw B, overwriting whatever was already in the frame buffer. The view now in the buffer is precisely the desired sampling of the visible portions of the scene in Figure 5a! This means that visibility determination for a number of faces can be done by finding a partial ordering under the above precedence relation, and drawing them out, one by one.

For two faces A and B, the possible relations are exhausted by:

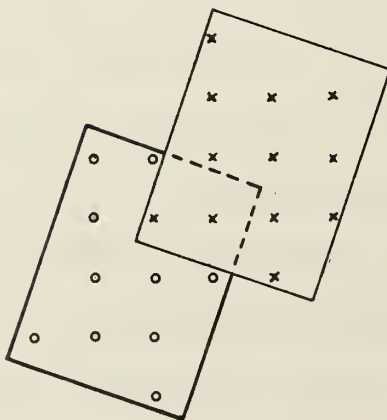
1. $A < B$ B occults A, A does not occult B.
2. $B < A$ A occults B, B does not occult A.



(a) Face A hidden by face B



(b) Face A drawn



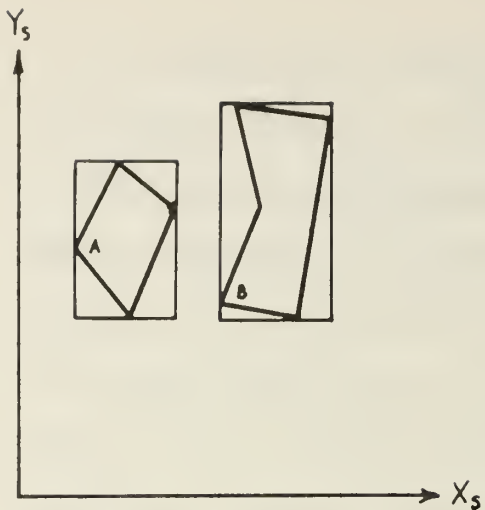
(c) Face B drawn, overwriting
face A

3. $A=B$ A does not occult B, B does not occult A.
4. $A?B$ A occults B, B occults A (cycle of length 2)

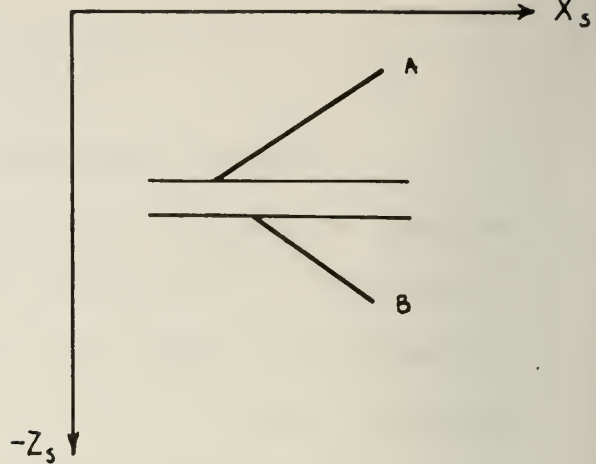
To determine the relationship of two faces, then, we need an algorithm that will answer the question: does A occult B? It turns out that it is much easier, and just as useful, to answer the question: does A not occult B? We may do this by applying a series of tests, the success of any implying that $A \leq B$ ($A=B$ or $A < B$). If any test fails, the next one is attempted. If they all fail, it is then known that either $A > B$ or $A ? B$. In this case, A and B can then be swapped, and the tests applied again to determine which is the case.

The tests described below are carried out in the screen coordinate system, and attempt to determine as rapidly as possible if A does not occult B.

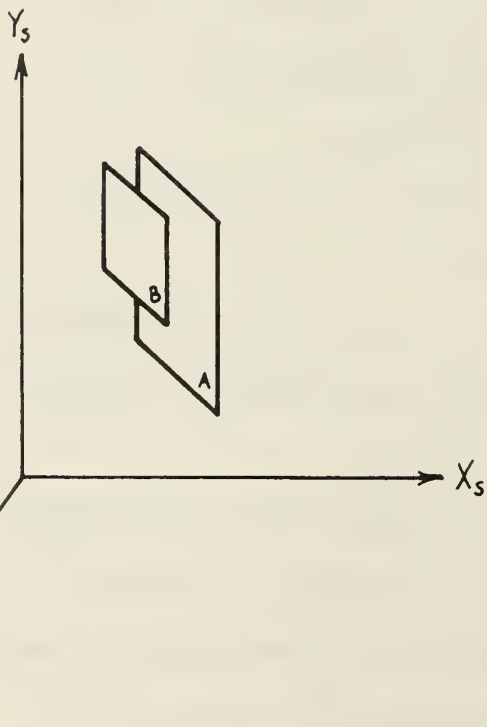
- Test 1. Maximum and minimum X_S and Y_S values of A and B do not overlap (Figure 6a).
- Test 2. Minimum Z_S coordinate of A \leq maximum Z_S coordinate of B (Increasing Z is away from observer) (Figure 6b).
- Test 3. No part of A is contained in the front half-space of B, or no part of B is contained in the back half-space of A (the plane of a face divides space into two open half-spaces and the plane of the face. The observer is contained in the front half-space) (Figure 6c).
- Test 4. (Every pair of edges E_A in A and E_B in B may have to be examined) We look for a pair such that if



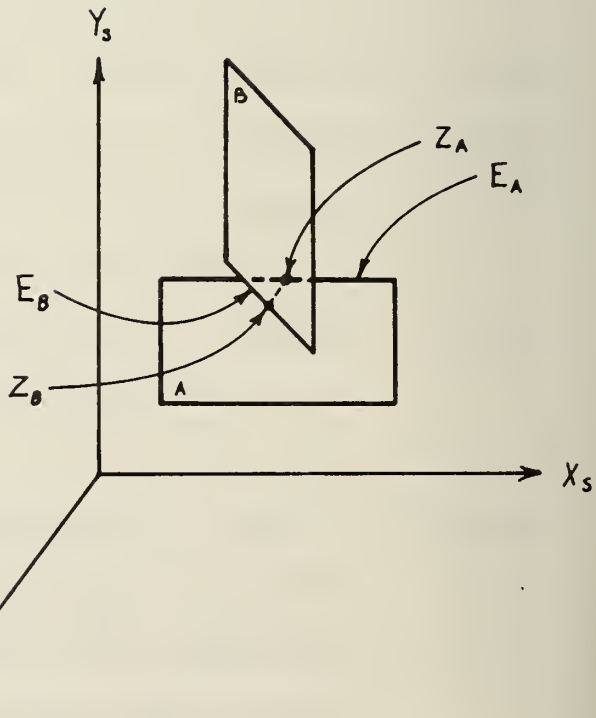
(a) Extreme values do not overlap



(b) A is not in front of B



(c) B in front half space of A



(d) B occults A

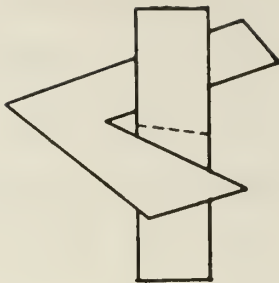
Figure 6. Comparison Tests for Precedence Determination

the projections of E_A and E_B intersect in exactly one point (in the $Z_S=0$ plane), then E_B is in front of E_A ($Z_B < Z_A$ in Figure 6d).

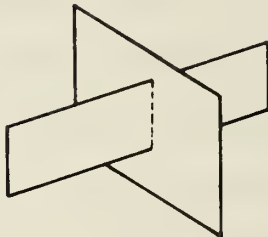
If the scene permits it, the relation \leq may be consistent and a partial ordering of the faces can then be determined by any (efficient) sorting algorithm. In fact, if the objects in the scene are separated enough, either Test 1 or Test 2 will always succeed; by either manually or automatically sorting their extents according to that rule, we can display several large objects, even though there is only enough computer memory to store the complete description of one such object at a time.

There is no such partial ordering only if there is a cycle of precedence: $A > B > \dots > A$. Cycles of length 2 are detectable when $A \neq B$, as in Figures 7a-b. A longer cycle is shown in Figure 7c. In all of these examples, we can cut the cycle and restore the existence of a partial ordering by splitting a face on the indicated dashed line. A special sorting method which detects such cycles and breaks them is needed.

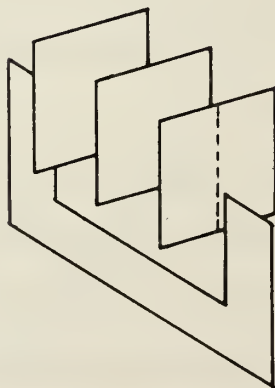
The basic idea used in the special sorter for cycle detection is to observe that there are no "minimum" faces in a cycle, faces with no other faces preceding them. Thus, we can detect cycles by searching for such a minimum face; failure indicates the presence of a cycle. It is clear now how the special sorter should work: a face P is selected as a possible minimum face candidate and compared



(a) Length 2



(b) Length 2



(c) Length 4

Figure 7. Cycles

with all other faces Q . If there is a face Q such that $Q < P$, Q becomes the new candidate--unless it had already been one. This can be determined if Q was marked when it was knocked out previously. The flow chart of Figure 8 illustrates this process as used in the special sorter (the parts in parentheses are comments).

The only problem remaining is determining the correct face to split. For length 2 cycles, the solution is to split one of the faces with the plane of the other face, as in Figures 7a-b. The proof of this is that it is now obvious that the pieces will pass Test 3, at most. For longer cycles, a special method has to be used. For example, in Figure 9, face P must be the one to be split, but suppose we discover the cycle when we compare Q and R ; how then do we find P ?

The approach we use is as follows. Let the cycle found be (1) $P > Q > R > \dots > P$. All of the relations are $>$ because the tests that were satisfied were: P occults Q , and Q does not occult P . Both P and Q are available because the cycle was detected when it was found that $P > Q$. The splitting process splits face Q into two sets of faces Q_1 and Q_2 for which we have $P \leq$ every face in Q_1 and $P >$ every face in Q_2 . Let us do that by projecting P onto Q and cutting Q at the projected edges of P (Figure 9). Note that the Q_2 portion is completely occulted by P . We clearly have here what is desired: $P \leq Q$, since P_1 does not occult Q_1 , and $P > Q_2$ since it does.

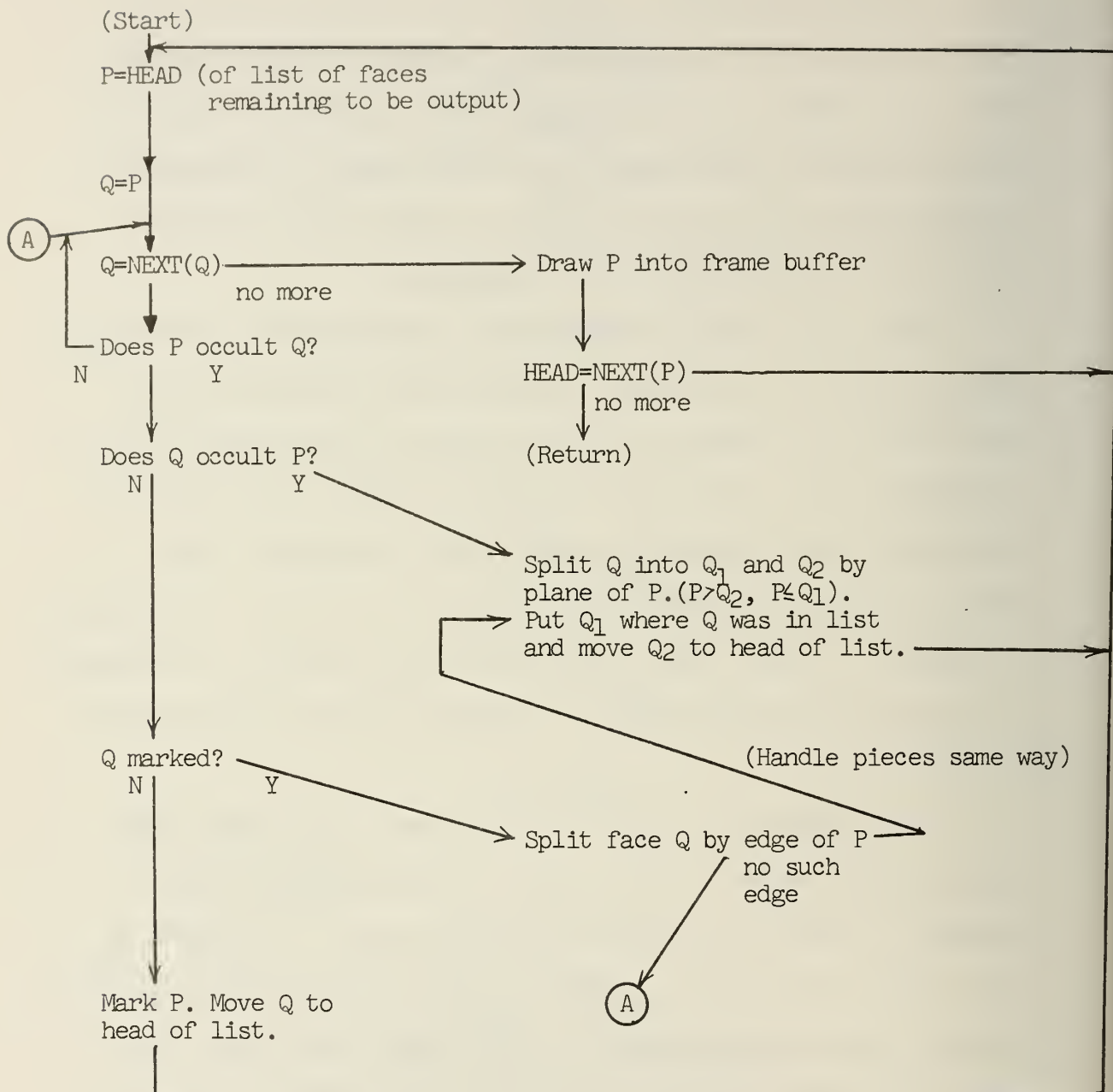
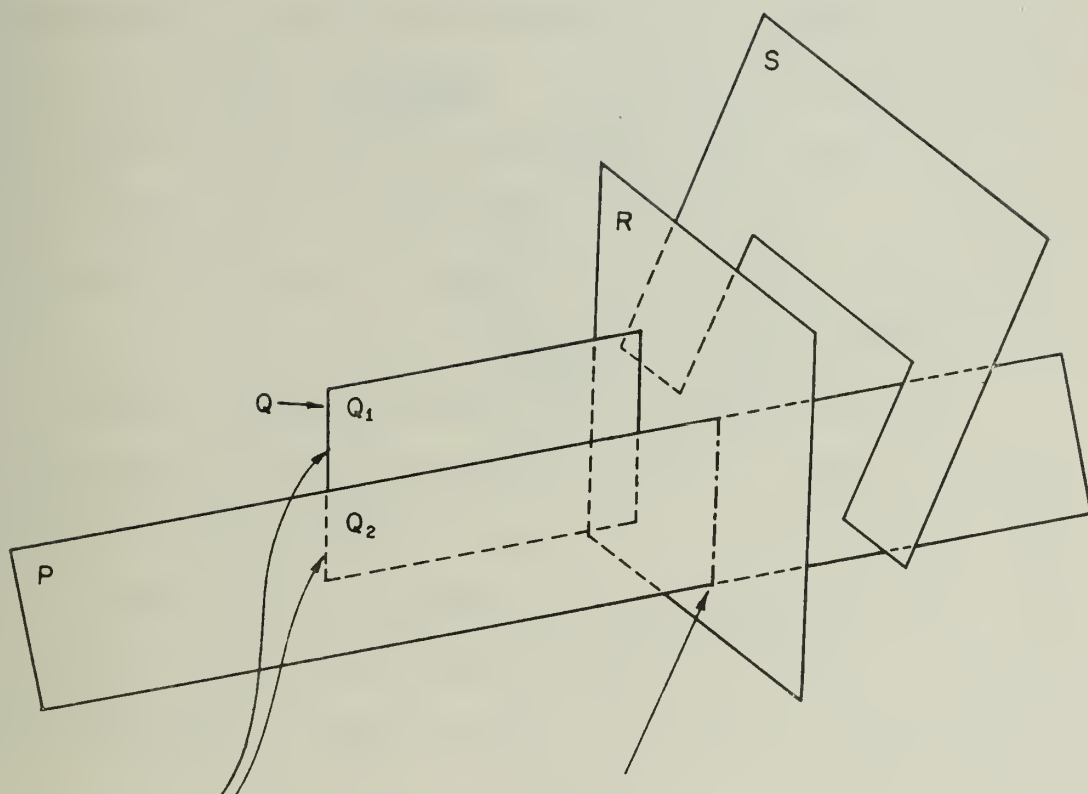


Figure 8. Special Sorter Flow Chart



Splitting Q results in
 $P \leq Q_1$, thus breaking the
 large cycle $P > Q$
 $> R > S > P$. However, P
 can still not be drawn
 unless it is split

here.

Figure 9. Multiple Cycles

Let us examine what happens to the cycle for the two parts of Q . Since $Q > R$, $Q_1 \geq R$, and (1) becomes (1)' $P < Q_1 \geq R > \dots > P$ which is no longer a cycle. With Q_2 , there are two possibilities: $Q_2 \leq R$ or $Q_2 > R$. The first case results in breaking the cycle as in (1)'; to handle the second, we must have arranged the special sorter so that the current cycle is the smallest cycle containing P . Assuming that has been done, we can see that $Q_2 > R$ cannot occur. Since Q_2 is wholly contained in P (in projection), $Q_2 > R$ implies that either $P > R$, $P < R$, or $P ? R$. In every case, smaller cycles containing P occur, the last being of length 2.

The method used in the special sorter to detect cycles in order of size is simply to look at an ordered list of marked faces first before looking at any unmarked ones. The marked faces are already in visible order: $T \leq S \leq R \leq Q \dots$, and the current minimum candidate is compared by examining the sequence, left to right. This method works, as can be proved by induction.

The fundamental stage in breaking long cycles is the projection of face P onto Q ; actually, we very much want to avoid doing this explicitly because that type of work is meant to be done implicitly in the drawing process. However, projecting just one edge of P onto Q can be done by the face splitting routine we already have; we just have to find an edge of P that intersects an edge of Q (as in Test 4. We use that routine, in fact) and use the plane of that edge to split Q (Figure 10). We then hope that that splitting

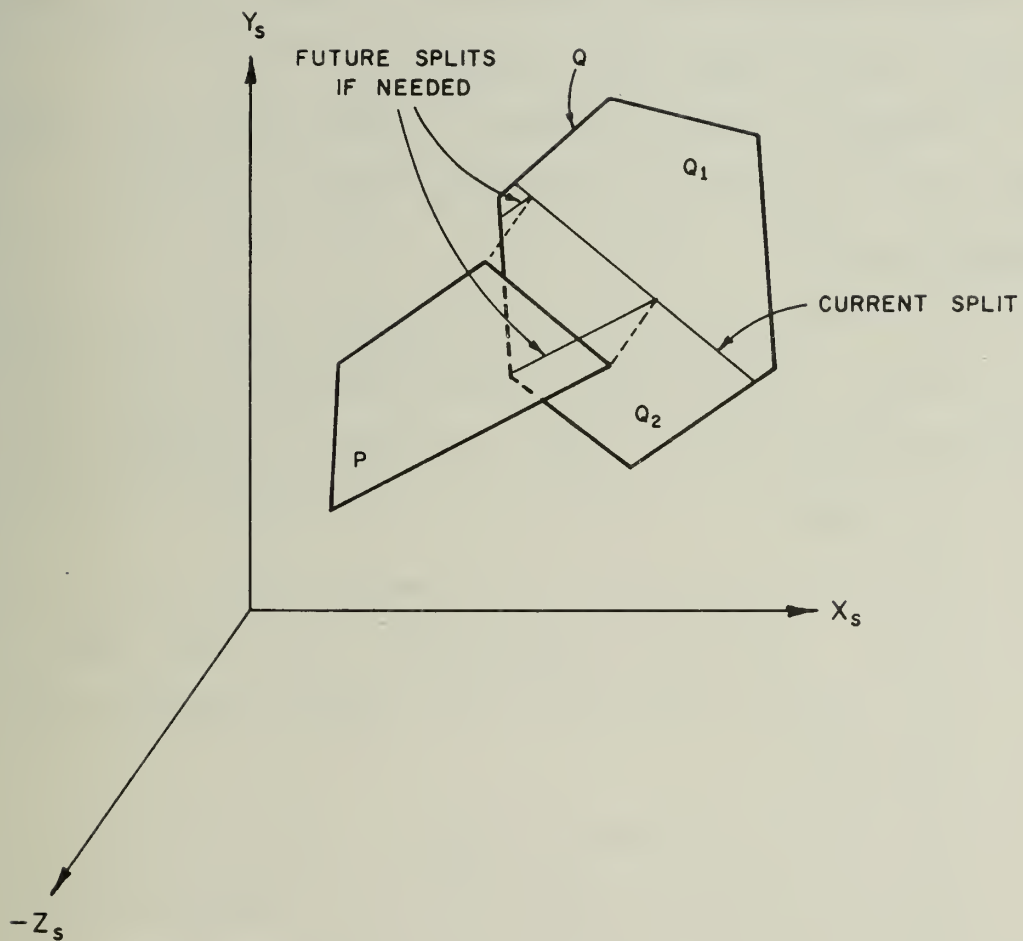


Figure 10. Projection of Edge Plane

was sufficient to break the cycle. That is a reasonable hope in general because most scenes have few cycles which are widely separated. If that splitting failed, another edge of P will be used (since the old one will no longer intersect Q in just one point), and we can see that, if needed, all edges of P will eventually be projected and Q will be divided correctly, and the proof above applies.


There is just one difficulty that may occur. If no edge of P intersects any edge of Q , there are three possibilities:

A. The projection of P is wholly contained in that of Q . In this case, we can use any edge of P to form the cutting plane. We always assume this is the case.

B. The projection of Q is wholly contained in that of P . But this is impossible, for Q would then be Q_2 , and $Q > R$ would lean $Q_2 > R$, which we showed could not occur.

C. P and Q overlap at edges or vertices only.

This is detectable when case A is assumed above, and the face splitter finds nothing to split. If all pairs of edges have been tried, P and Q do not really overlap, and we reverse the decision of test 4. Otherwise, we must try another pair of edges.

As an interjection, it is interesting to note that the comparison routine described above can be effectively used when manually drawing line drawings of three dimensional scenes. In Figure 11, one wants to draw the 3D block letter E. Because of the face ordering, one knows that  can be drawn first. The rest of the faces are easily taken care of, as in the sequence. In fact, this method is not restricted to manual means; a special drawing routine can be used to produce a line drawing (on a plotter) of each face. This routine keeps track of the current "visible perimeter," and when a face is given to it to be drawn, only the part outside the visible perimeter is plotted, and the perimeter is then updated. The major effort of the routine is spent on determining whether or not the face is inside.

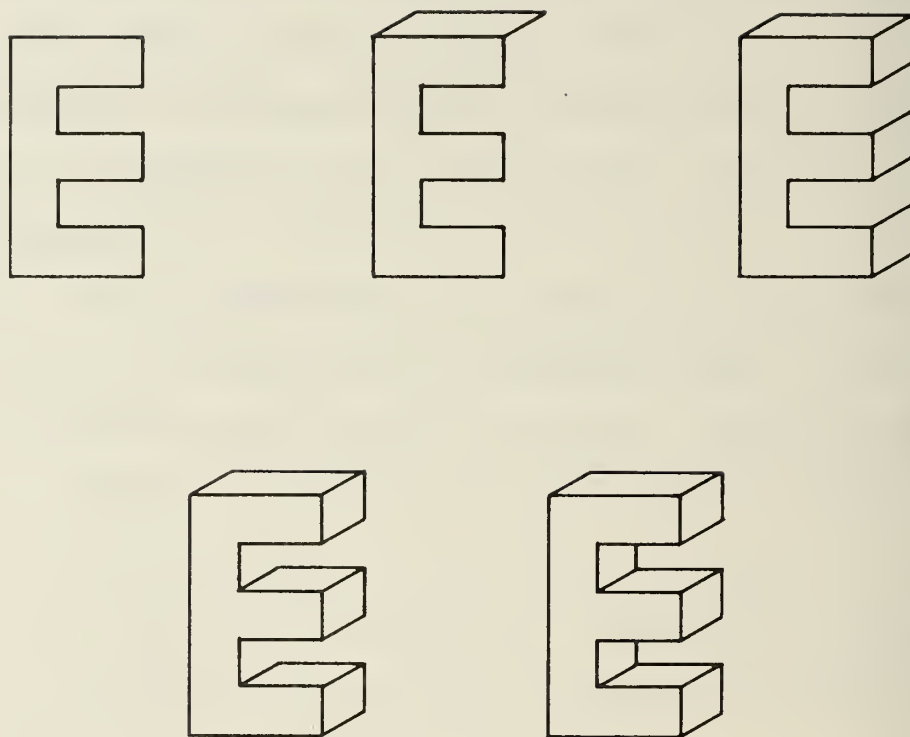
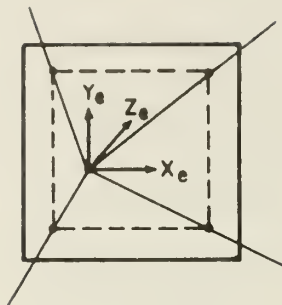


Figure 11. A Manual Algorithm for Drawing
3D Objects

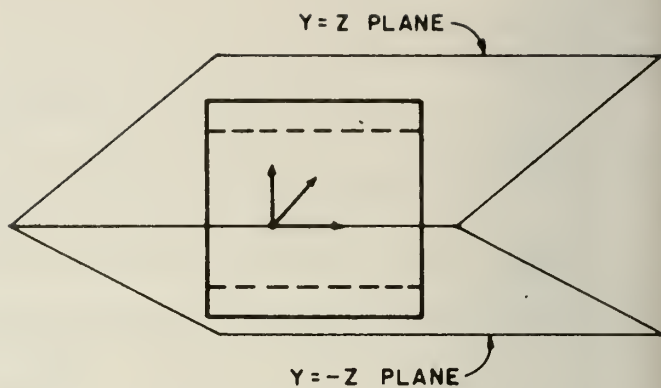
4. FACE SPLITTING

The special sorter requires an algorithm for splitting faces. Furthermore, in the transformation from world to screen coordinates, polygons and polygon portions that lie outside the viewing pyramid (Figure 3c) must be removed (or "clipped") from consideration, because these parts are not going to be visible. An efficient edge clipping routine already exists [9], but it cannot be directly applied to polygons. As in Figure 12a, all of the edges of the face will be rejected as being outside, but since the face surrounds the viewing pyramid, it is certainly potentially visible. Edges must then be generated for the clipped face, as in the dashed part of the figure. Our approach to face or polygon clipping is to use the face splitting algorithm to split faces with the planes that define the sides of the viewing pyramid. For example, the face of Figure 12a can be split with the planes $Y=\pm Z$, resulting in 2 edges being generated and the polygon of Figure 12b. (Each edge can simultaneously be compared with both planes, so we can do it in one pass.) Finally, the resulting face(s) are split using the planes $X=\pm Z$, and it is seen that the desired result is necessarily obtained (Figure 12c) (Of course, we throw away the parts outside).

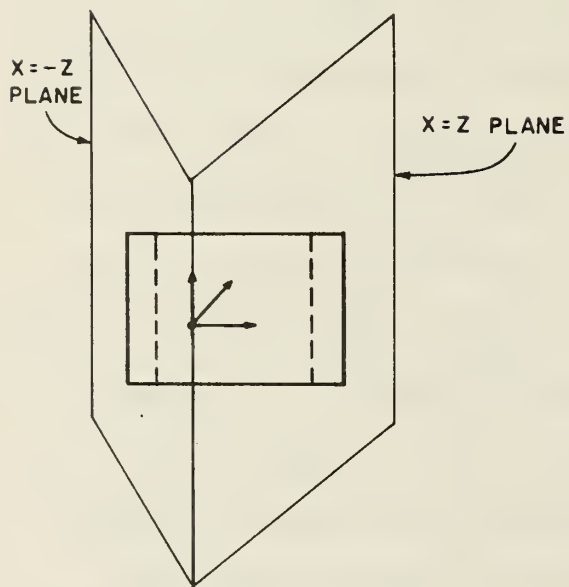
The point of that discussion was to show that the face splitting algorithm can be used either by the special



(a) All edges outside
(no edges clipped)



(b) 2 pass polygon clipping: pass 1



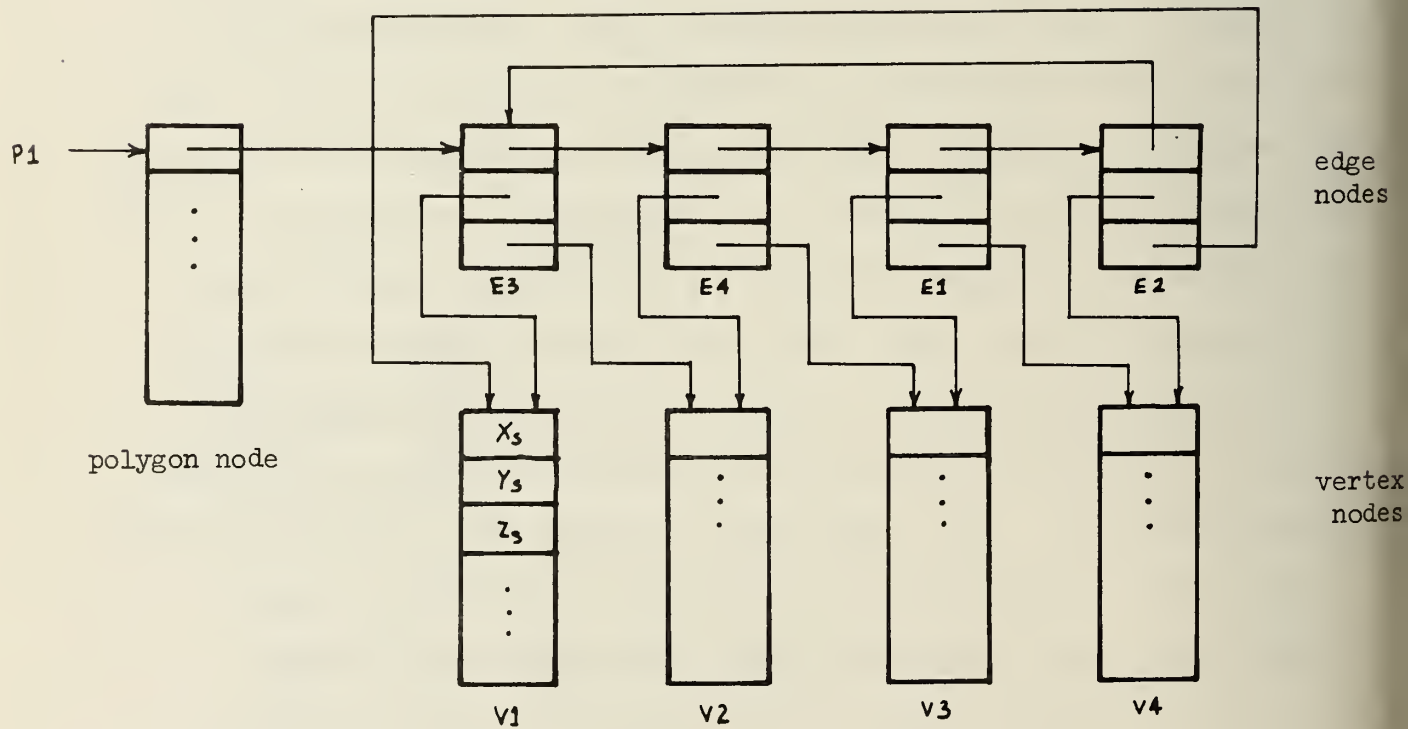
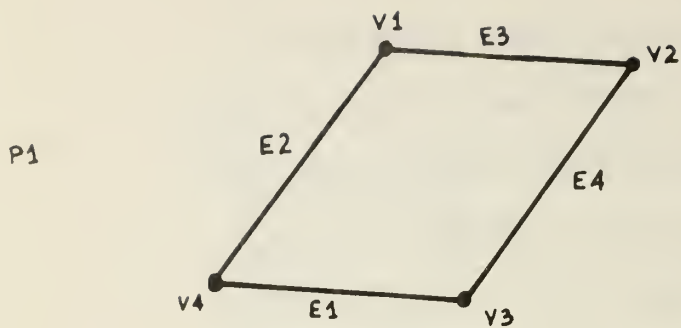
(c) pass 2

Figure 12. Polygon Clipping

sorter or the polygon clipper. Since polygon clipping is a necessary adjunct for any three-dimensional visible surface display system that can show close up views, the algorithm we describe can be used elsewhere, in such a system.

The method is given a polygon and a cutting plane, and divides the polygon into parts, also polygons, that lie to either side of the cutting plane. The polygon is assumed to be planar, so that the intersection points of its edges with the cutting plane lie on a line. It will work on twisted polygons if that assumption remains true. The structure of the splitting algorithm is affected by the data structure used to represent polygons. We have found that representing each polygon as a ring of edges is sufficient for our purposes. The edges are arranged in the ring so that the "head" vertex of one edge is the "tail" vertex of the next edge in the ring, and the polygon node has a pointer to some edge on the ring. Figure 13 shows a polygon and its representation in the terms described above. The requirement is then that the face splitter return the face fragments resulting as a list of polygon nodes with their associated edge rings and vertex nodes.

It becomes clear how the splitting algorithm should work. First, all of the edges on the splitted face must be examined. Now, there are four possible ways the endpoints of these edges can be related to the cutting plane: both may be in the same halfspace, both may lie in the cutting plane, exactly one may lie in the cutting plane, or they may be in different halfspaces. The contrived polygon of



V1 is tail vertex and V2 head vertex
of E3

Figure 13. Polygon Data Structure

Figure 14a shows all four types; the edges have been labelled with letters A, B, C, or D, respectively, corresponding to these four ways. Ignoring type B edges for the moment, and after splitting each type D edge into two type C edges, it is clear that the algorithm need deal only with type C edges. At this point, then, we have just an unordered list of type C edges; if we can structure this list into a representation of Figure 14b, we will be nearly done, because that structure has all the information needed to complete the splitting process. Deferring the justification, observation shows that the conversion from unordered list can be done in two steps:(parenthetical comments refer to Figure 14b):

1. Generate two lists of edges (the "left" and "right" lists); each list contains edges all in the same half plane (so the left list contains edges 1, 3, 6 and 7, and the right list edges 4, 5, 8, 9, 10, and 11).
2. Sort the edges of each list with respect to the position of their endpoint on the line of intersection. Then, if A comes after B and B comes after C on the sorted list, the endpoint of edge B lies between those of A and C on the line of intersection (sorted left list becomes 1, 7, 6, 3 in order, and right list 11, 10, 9, 8, 5, 4).

The splitting process can now be completed as follows:

3. For the right, say, then left lists, we take consecutive pairs of edges and join them with a generated edge. (Thus, 11-10, 9-8, 5-4, 1-7, and 6-3 are edges that will be joined with new edges, which are dashed.)
4. After all pairs have been joined, we observe that we now have 2 or more rings of edges, each with at least one generated edge on it. We have kept the generated edges from step 3 on a list, and we now pick up each edge on it and mark all edges in the ring containing it. We continue to do this for each unmarked edge remaining in the list of generated edges. For each ring found in that process, we generate a polygon node pointing to it and append that node to the output list of polygons. (Thus, the rings found will be 1 2 3-6 7-1, 4 5-4, 8 9-8, and 10 11-10.)

Step 3 is clearly the crucial step because it is not obvious that the obtained edge pair can be joined. Edges A and B in Figure 14c cannot be joined by an edge and made part of a ring, since either edge A or B would point back at the joining edge. We will show that if the polygon being split is planar, closed, and non-self-intersecting, the scenario of Figure 14c or any similar contretemps cannot occur.

Given such a polygon, the intersection points of it with the cutting plane lie on a line; the property we wish

to prove is the alternation property, as illustrated in Figure 14d. Travelling along the line of intersection from A to B, the polygon edges encountered on both the left and right side alternate in the direction they point: the endpoints lying on the line of intersection alternate as the head, tail, head, tail, ... (or tail, head, tail, head, ...) vertex of the edge. That this must be so can be shown figuratively: there is a one armed man at C in the figure, and he walks around the polygon in the same direction always, so that his right arm points inside the polygon. Since it is closed and non-self-intersecting, this is always possible. Consider then a detail of the line of intersection, shown in Figure 14e. The line can be divided into segments, which are alternately inside and outside the polygon, by Jordan's theorem from topology. If the segments are as described in the figure, the man will cross at edge A in the direction of the arrow, with his right arm pointing to the inside segment. Therefore, edge B, the very next edge, must cross the line of intersection in the opposite direction; otherwise, the man's right arm, when crossing at that point, would point outside. The same can be verified for edges C and D. Thus, the direction always alternates as desired. Finally, since the extreme ends of the line of intersection are necessarily outside, there will always be an even number of type C edges intersecting, by Jordan's theorem again. Since there are an even number of crossings (edges A, B, and D in Figure 13e), and each edge of type C counts as two intersections, both the left

(top) list and the right (bottom) list contain even number of edges. Therefore, step 3 in the algorithm will never have difficulties, and the algorithm is correct.

There is one problem resulting from the implementation of the algorithm, and it concerns step 2, the sorting step. Rather than sort along the line, it is much faster to sort along the projection of the line; for example, in Figure 13a, it is obvious that sorting along the X_s axis, i.e., sorting just by the X_s coordinate of the intersection, is equivalent to sorting along the line. If the line of intersection happens to be parallel to the Z_s axis, for example, we cannot sort by X_s any more, but then we can use the Z_s coordinate instead. Sorting by this method, then, edges 9 and 10 of Figure 13b, since they have the same endpoint on the line and are both in the right list, are not necessarily sorted in correct order; the ordering 11, 9, 10, 8, 5, 4 may occur. Since the algorithm is known to work, we can handle this case by watching for failure in the joining; linking edges 11 and 9 will fail, and this must mean that edge 9 and the next edge 10, are in the wrong order. They can then be switched in the ordered list, and the joining continued.

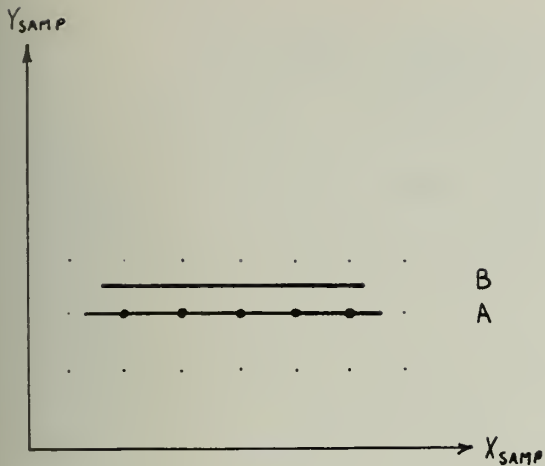
One last remark should be made. Polygons that lie completely in the cutting plane will disappear with this algorithm; however, this does not cause any change of the display, because such a situation can only occur if the polygon is edge-on to the eye, and thus invisible anyway.

5. FACE SAMPLING

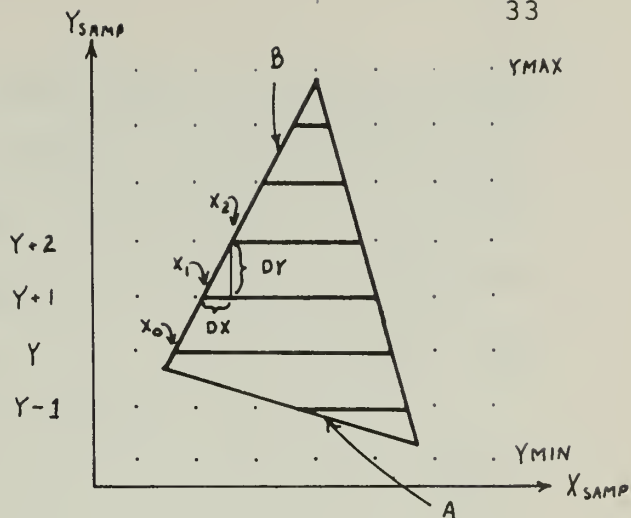
Every time a minimum face is found by the special sorter, the visible surface determination algorithm requires that the locations of the sample points representing that face be found (recall Figure 5), and the raster frame buffer then overwritten at these points with the intensity value of the corresponding points on the face.

Sample determination is not affected by the shading method used, so we discuss the shading algorithm separately, in the next chapter.

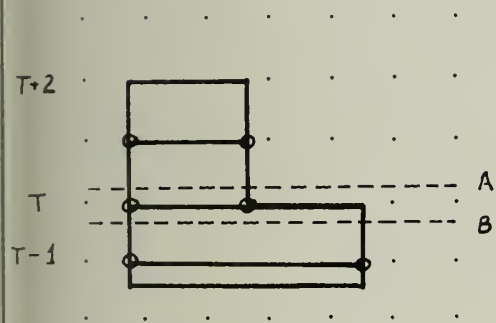
Our approach to determining the sampling of a face can be understood by observing a much simpler case. In Figure 15a, line segment A lies on a scan line ($Y_{\text{samp}} = \text{integer}$). Clearly, the sample points on that line are the points where its X_{samp} coordinates are integers. The sampling problem for line segment A is thus trivial. If another similar line segment parallel to the X_{samp} axis, but not with an integral Y_{samp} value exists (line B), it is just as obvious that no sample points occur on that line. We can then reduce the sampling problem to determining the intersection of the Y scan lines, say, with the face being sampled. The resulting "segment" representation of the face is shown in Figure 15b. If the scan lines are examined consecutively, the necessary problem of determination of segment endpoints can be simplified. These desired endpoints clearly lie on edges of the face; thus



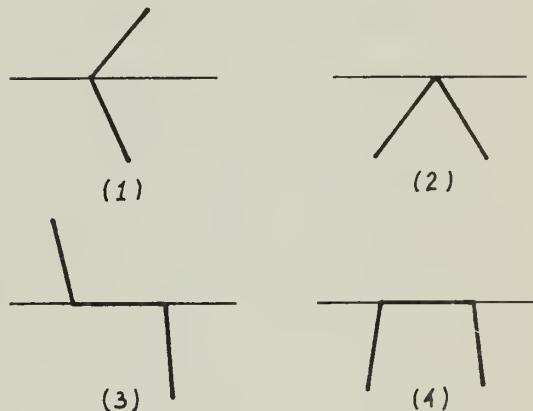
(a) Simple sampling problem



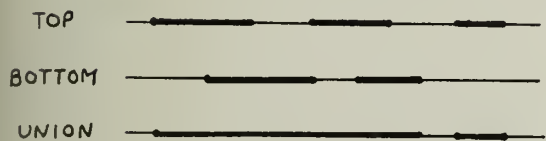
(b) Segment representation



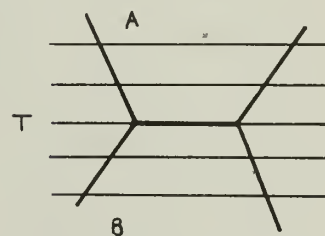
(c) Segmentation errors



(d) Possible edge configurations when vertices lie on scan lines



(e) Segment union operation



(f) Sampling convention

Figure 15. The Sampling Problem

the X_{samp} 's of the intercepts of a particular edge are necessarily linearly related. As in the figure, the left endpoint of the segment at line Y is X_0 ; since DY is 1, X_1 is then related to X_0 by $X_1 = X_0 + DX$ and we must have (1) $X_n = X_{n-1} + DX$ ($n = 1, 2, \dots$).

Now if no vertices of the face lie on any scan line, we know there will always be an even number of edge intersection points on every scan line, because the extreme points of the line are necessarily outside the face. Thus, we can sort the points of intersection along the scan line, and use consecutive endpoints as marking the desired segments. This idea is also used in the face splitter, where the sorting is done along the line of intersection of the face and the cutting plane. Thus, a working sampling algorithm would do the following things consecutively for each scan line:

- Update, using (1), the position of the X_{samp} intercepts.
- Remove from consideration edges no longer intersecting. (In Figure 15b, edge A does not intersect scan line Y and thus must be deleted when going from line Y-1 to Y.)
- Compute the X_0 values for edges being considered for the first time. (Edge B in the figure appears initially at line Y.)
- Sort the X_{samp} intersects.
- Output each consecutive pair of endpoints as segments (the shading algorithm is called to shade the segment).

Only one problem remains. In Figure 15c, some vertices of the face this time lie on scan lines; the circles mark the segments found by the above method. On line T, the segment is clearly too small, and no segment at all appears on line T+2, even though one should. The cause of this difficulty is that a vertex lying on a scan line can either represent one or two intersections, depending upon the configuration of the edges coming from it. Four possibilities are represented in Figure 15d; only for case 1 is the algorithm correct. The vertex in case 2 and the horizontal portions in cases 3 and 4 are not always output.

One method for handling this problem requires two "passes" for each scan line with a vertex on it. Its motivation derives from the fact that the problem above can always be made to disappear by perturbing the scan line (very) slightly upward or downward, keeping it parallel to the X_{samp} axis. If the segments found differ, their union is used as the actual segmentation of that scan line (Figure 15e). As in Figure 15c, the segmentation of the perturbed scan lines A and B differ (parts between X's); their union is identical to the segmentation on B, which is clearly the correct one. In line T+2 of the same figure, there will be no segments on one perturbed scan line, but the other, and thus the union, will be as desired. The proof of this method is to note that, on the scan line concerned, a point is in a segment iff it is in one on either the top or bottom perturbed line; thus, iff it is in their union. Implementation of this

method requires a segment buffer, where information on segments generated for the current scan line is accumulated. This can be done by adding another step to the sampling algorithm. Before deleting exiting edges (in the second step), a check can be made to see if any of these edges had an endpoint exactly on the current scan line. If so, then segments are output as in the fourth and fifth steps before these edges are deleted. The normal output step in step 5 now can just overwrite this segment buffer to gain the desired union. It can be seen that our face-splitting algorithm uses the same idea when it uses the "left" and "right" lists separately to determine what edges (i.e., segments) to generate for the face.

Further reflection on this problem results in the fact that the error it causes is at most one scan line wide (e.g., line T+2 in Figure 15c). If the special shading algorithm is used, each output scan line is actually the average of N scan lines (N=4, usually) covering the same space in greater detail, and the effect of the missing line or line part is reduced by this factor. Furthermore, in a case like that of Figure 15f, where two faces are shown, scan line T should actually have an intensity which is the average of face A and face B's. The result of the error above is that the line has just A's intensity. Thus, what we really have here is a convention, expressed by saying that the last scan line an edge appears on is $Y_{LAST} = \lceil Y_{MAX} - 1 \rceil$ ($\lceil \cdot \rceil$ represents the ceiling function, and YMAX is the maximum

Y_{samp} value of the edge). So, if YMAX is not an integer, say, 12.37, YLAST is $\lceil 11.37 \rceil = 12$, as desired. If it is, say, 12.00, then YLAST is $\lceil 11.00 \rceil = 11$. The same type of convention is applied in the shader to the left edge of segments. More information on that is found in the next chapter.

Using that convention, then, we can "ignore" the problem and maintain the simplicity of the face sampler.

It is of interest here to discuss briefly two techniques for speeding up the face sampler. First, the edges of the face can be stored in a list which is initially sorted by the minimum Y_{samp} value of the edges; this allows us to immediately determine for each scan line whether or not a new edge appears. Since there are few edges in general on a face, a merge insertion sort is acceptably fast for creating this list. Note that this technique can also be applied to the list of faces used in the special sorter; we can have sorted that list initially by the maximum Z_s coordinate of the face. Thus, we can stop comparing the current minimum face P if it is found that minimum Z_s coordinate of P $<$ maximum Z_s coordinate of the next face Q on the list of faces; since all further faces have maximum $Z_s \geq$ that of Q, we know that P cannot occult any of these faces (recall test 2) and that then it must actually be the desired minimum face.

The second technique seems applicable only in the face sampler of all of our algorithms; we discuss it nevertheless in case we are wrong. In the sampling algorithm, the current scan line intercepts are sorted every time; actually, this

is not necessary because the relative position of an edge intercept with respect to the others does not change. (If they did, it would mean that a pair of edges intersected not at a vertex, which cannot happen in a projection of a planar polygon--unless it is edge-on. But in that case, it is not visible anyway and we need not consider it at all.) Thus, we can maintain an X-sorted list of edge intercepts; when new edges are found, we need only merge them into that list.

6. SEGMENT SHADING

The face sampling algorithm determines the segments of the current scan line that lie on the face; it now becomes the shader's task to take these segments and for every sample point on them determine the intensity value to use to represent that portion of the face. For an exact intensity rendition, it is clear that we would have to consider the texture, transmittivity, translucency, the scattering, specular, and internal components of reflection of the object, light reflected and shadows caused by other objects, position, size, and spectra of light sources and position of the eye. Rather than attempt such a precision, we concern ourselves only with producing an acceptable rendition by choosing just one or two of these intensity effects to model in the shader. Figure 4, for example, was shaded using only a scattered shading model. Furthermore, intensity is also affected by sampling error correction; as in Figure 2c, the intensities of the neighboring sample points of A must be modified to indicate A's presence. In this chapter, then, we will discuss only the scattering, specular, and transparency models. Once the appropriate shade is obtained, we show how to use these values to reduce the effect of sampling error.

For the shading models, only one white light source is considered, and it emanates parallel rays of light. If this light source is placed at the eye, it is clear that no

shadows can be seen (if all surfaces are opaque). Historically, then, this was done to avoid the computation of shadows. Newell [1] seems to have been the first to discover that a shadow-like effect results from the scattering model if the light source is somewhat displaced from the eye, resulting in much more interesting displays for certain classes of objects. The other previous efforts in visible surface display [2-8] also have slightly differing shading rationales.

As they have done, we compute the intensity of a face just at one point, say, its centroid, and use this value over the entire face. The shading models that are global in this way are the scattering and specular reflection off the face; transparency must be handled on a local basis because it depends on what faces are in back of the current one. We do not cover any other models.

Scattering reflection is embodied in the following intensity rule:

$$(1) \quad I = I_0 \cos^a \theta$$

where θ is the angle between the face normal and the parallel rays of light. a is a small value, 1 say, and models the scattering component. This component and its near independence of the eye position can be demonstrated by a simple experiment. Take a pad of white scratch paper and extinguish all room lights except the fluorescent desk light. By rotating the pad of paper around an axis parallel to that of the lamp, and by moving the head, an intensity variation similar to

$I_0 \cos \theta$ is observed. I_0 is proportional to the intensity of the light source and the reflectance of the face.

Specular reflection models the mirror-like qualities of the face; it is non-existent for paper, but appears in shiny or metallic surfaces. It is not independent of the eye position, since we are actually seeing the reflection of the light source. It can be modelled by

$$(2) \quad I = I_1 \cos^b(\theta - \theta_1)$$

where θ_1 is the (absolute value of) the angle between the face normal and the face centroid to eye vector. b here is a high value, 5-20, because reflections only appear at very narrow critical angles. I_1 is again proportional to the reflectance and the source illuminance. If the surface is ground metal, for example, both scattering and specular components will appear, and the sum of (1) and (2) can then be used to model its intensity. Figure 16 shows a face and all of the angles used in the above two computations.

Finally, the face transparency is computed at every sample point by comparing the intensity given by (1) and (2) above (I_a) with the intensity at that point already in the frame buffer (I_b). The output intensity I is

$$(3) \quad I = I_a \quad \text{if} \quad I_a > I_b$$

$$I = (1-w)I_a + wI_b \quad \text{otherwise}$$

where w is proportional to the transmittivity of the face.

Thus, $w = 0$ means that the face is opaque, whereas $w = 1$

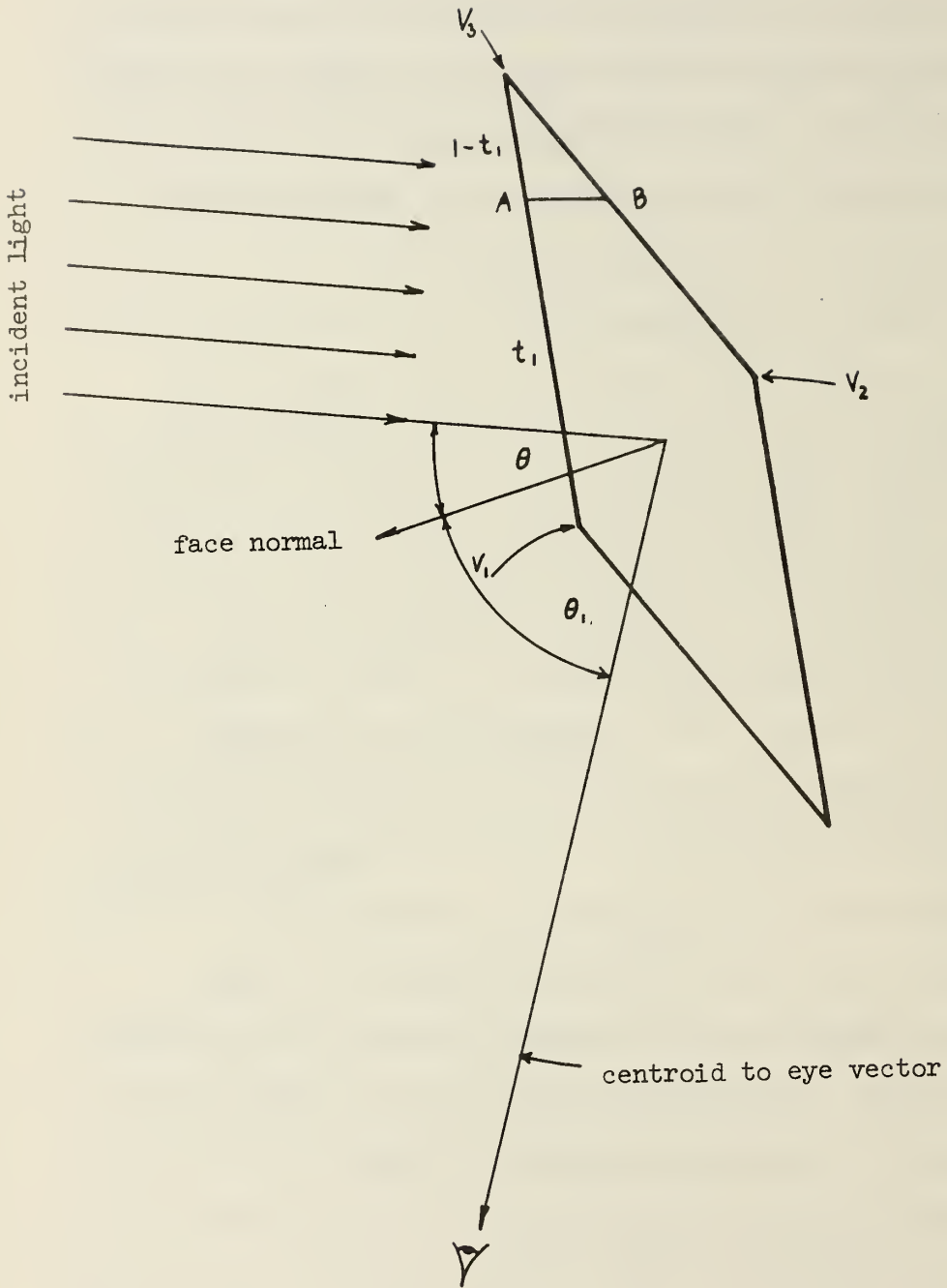


Figure 16. Illuminance Determination

means that it is completely transparent. The first part of (3) is used because physical observations show that it is hard to see through brightly-reflecting glass!

The shader can also be modified slightly to use Gourad's smooth shading technique [2], especially for displaying curved surfaces. These curved surfaces are still approximated by polygons, but the shading is so computed that there are no discontinuities in its value. Thus, "joints" between polygons that appear because of differing intensities are made to disappear. The shading method used is a double linear interpolation. First, normals to the surface at vertices are computed; usually, the average normal of the faces surrounding that vertex is used to approximate that value. An exact value for the normal is available, for example, when using Bezier or Coons-type surface representations. Then, the intensity of that vertex is calculated using just the scattering and specular models. Now, when a face is drawn out, the intensity of its edges is taken to be a linear interpolation of the intensities of their endpoints. As in Figure 16, the intensity of point A is effectively $t_1 \cdot \text{intensity at } V_1 + (1-t_1) \cdot \text{intensity at } V_3$, and similarly for B. The segment AB thus has intensity values for its endpoints, and the same type of interpolation is performed for every sample point along that segment (note that both interpolations can be done incrementally). It can be seen then that the major effect of that rule is to maintain continuity of intensity across the edges of faces.

The values obtained are finally modified by the transparency model as in (3) above. It should be noted that effective smooth shading requires enough intensity levels to avoid any kind of "contouring" effect, where a jump of one intensity level becomes immediately apparent. However, a sufficient number of intensity levels can always be simulated using averaging or pseudo-random coding [10]; for example, if an intensity of 10.75 were desired, we could output 10 1/4 of the time and 11 3/4 of the time, so that the average intensity output is 10.75. Use of a fast pseudo-random number generator to select the proper intensity level is preferable. If, Gourad's method is used, then the face splitting routine, when it splits type D edges, should compute a correct intensity value for the vertex it creates.

We are now ready to discuss the actual details of the shading algorithm. It should be clear now that the basic sequence of action is:

- Read scan line and do other initialization.
- From left sample point to right sample point:
compute either a constant or linearly-varying intensity value; apply transparency model; use pseudo-random coding to convert to an integral intensity value; and output the new pixel values.
- Write back scan line to raster frame buffer.

The (purposely) detailed flow chart of Figure 17 performs these functions. The significance of N will be covered later; assume $N = 1$ for the moment. The array g_0 is a buffer

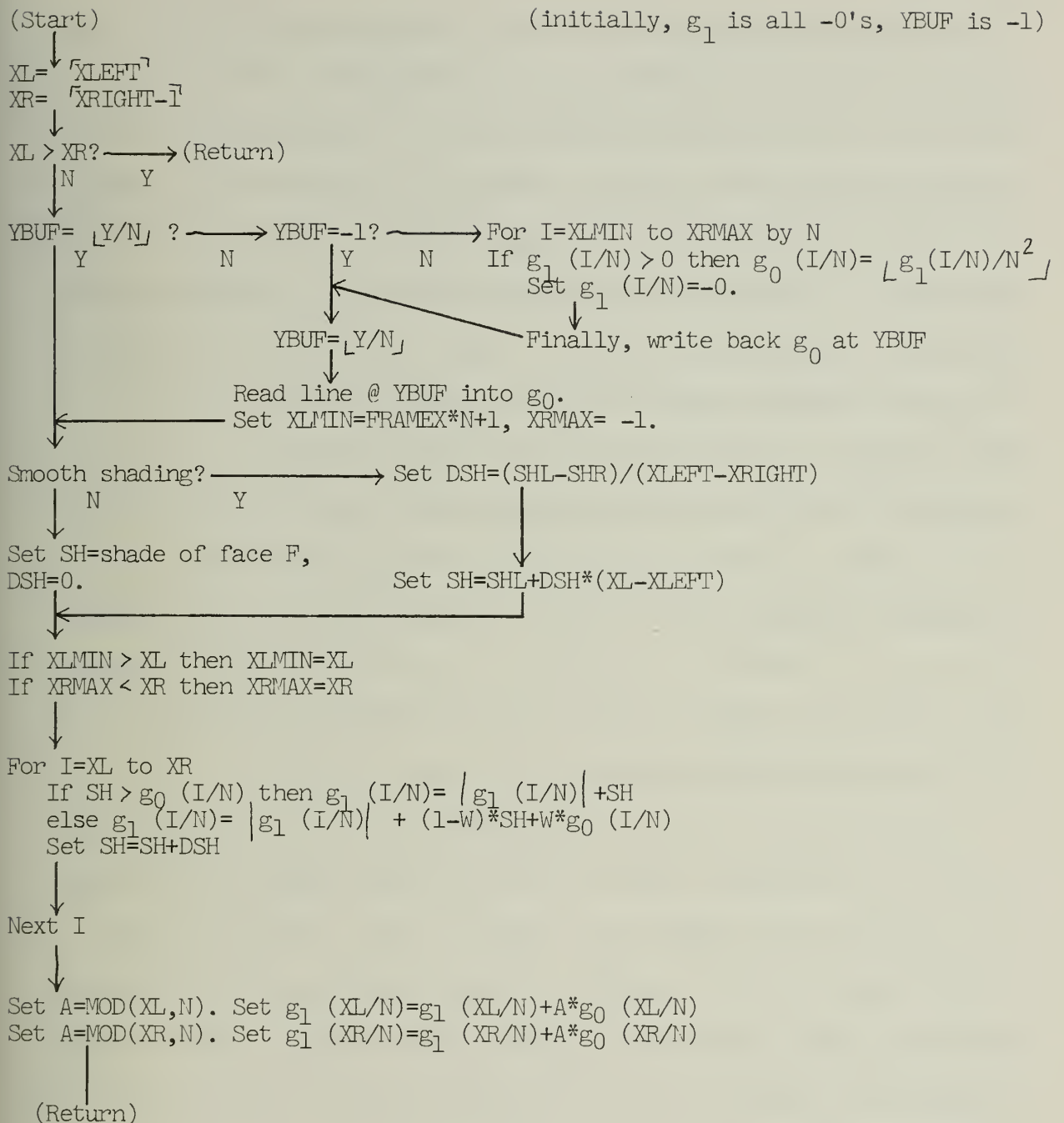


Figure 17. Special Shader Flow Chart

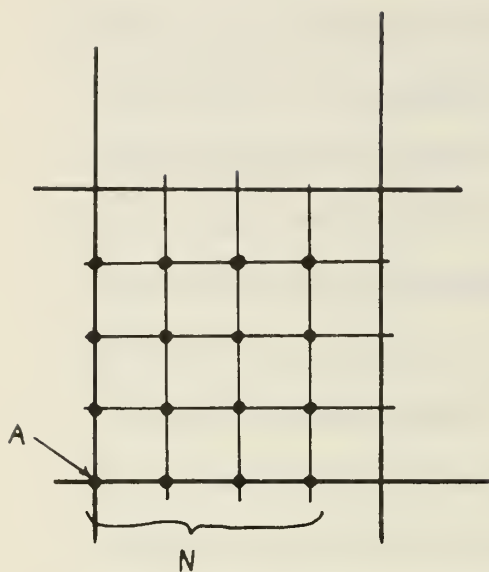
containing the pixel values of the scan line located at YBUF; when a new line is needed, g_0 is written out, and the new line read into g_0 and YBUF updated. The array g_0 is used to buffer the new pixel values; before g_0 is written out, the parts of g_1 that changed are copied to g_0 . The smooth shading technique is implemented incrementally by $SH = SH + DSH$, where SH is the intensity or shade to use. XLEFT and XRIGHT are the exact segment endpoints on the current scan line at Y; SHL and SHR are the intensities at the endpoints. If a constant intensity is desired, the shade of the entire face F can be used. XL and XR are the (integral) leftmost and rightmost sample points on the current segment. Note that we use $XR = \lceil XRIGHT - 1 \rceil$ instead of the expected $XR = \lfloor XRIGHT \rfloor$; this represents the use of the same convention mentioned when discussing the face sampler.

It is time to discuss our method for reducing sampling error. It should be clear now that we can reduce that error by increasing the number of samples; i.e., increasing the "resolution." But we cannot forever increase the resolution because of the finite size of the frame buffer. When maximum resolution is reached, we must look to other means of reducing sampling error.

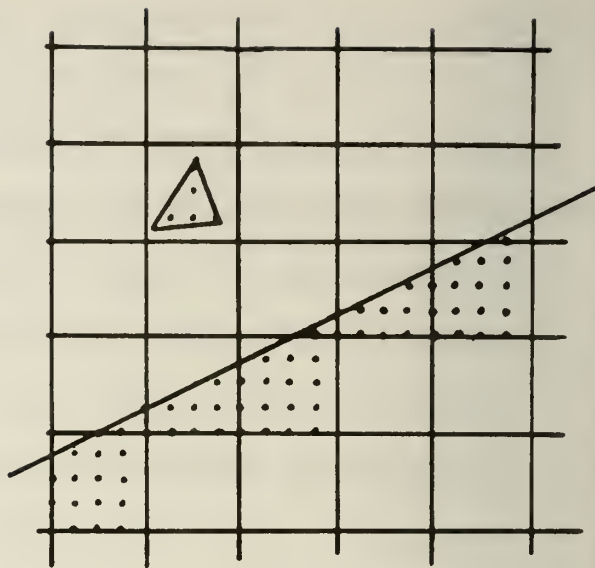
Let the maximum sample grid size be K^2 . Our special shading algorithm requires that samples be computed as if for a $(NK)^2$ grid; then this information is transformed (reduced) so that it fits into K^2 samples. The simplest such transformation that is of any use is to effectively divide the $(NK)^2$

grid into K^2 $N \times N$ blocks. Then, the average intensity value of each block is computed and output as one of the corresponding K^2 sample points. Clearly, more information is contained in this new K^2 grid as opposed to using just K^2 samples; as in Figure 18b, we have now become aware of figures that formerly lay between scan lines, and something has certainly happened to the jagged edges. Since the averaging transformation above is essentially a "blurring" transformation or a "low-pass filter," it is clear that at worst the sharply-defined jagged edges have become less-obvious blurred jagged edges. Other reduction transformations that lose less information may exist, and it should be an interesting task to find them.

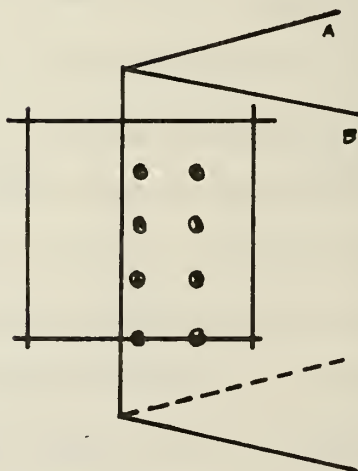
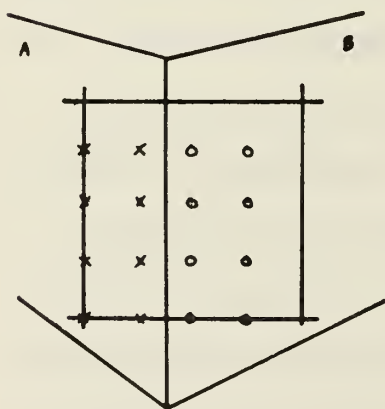
Modifications to the basic shading algorithm above required to implement the averaging are described now. Segments are found in the $(NK)^2$ grid, and the g_1 array is used to accumulate the intensity values. When each K^2 scan line is complete, after N steps, the elements in g_1 are divided by N^2 and then output. Figure 17 is also the flow chart of the special shading algorithm; the N referred to in it is the N we have been using. Figure 18a shows the exact position of the N^2 block; sample point A is set to the sum of the pixel values at the circled spots divided by N^2 . Thus, the transformation from scan line I in the $(NK)^2$ grid to the corresponding N^2 line is $\lfloor I/N \rfloor$, or just the quotient of I/N (integer divide).



(a) Placement of block



(b) Effect on small faces and jagged edges



(c) Demonstration of information loss - both have same average intensity

Figure 18. Sampling Error Reduction

Note that the speed of the shading algorithm using the averaging transformation is proportional to N ; the face sampler also has N times more scan lines to process, so the total effort is slowed down by a factor of N^2 . Actually, though, that can be cut to only N times slower by doing special-case handling of the inner loop in the shader; for long segments, we can effectively do the sampling of the inner part N samples at a time. Since the segment on the inside spans the block, we can compute what is to be added; we have the sum as

$$SH + SH+DSH + \dots + SH+(N-1) \cdot DSH = N \cdot SH + \frac{N(N-1)}{2} \cdot DSH$$

directly. Thus, for large faces, the shader will not run significantly slower. Other transformations may be implementable even more rapidly.

Actually, the above shading algorithm is preferably used with a Watkins-type visible surface algorithm that knows exactly what is visible at each sampling point. Since each face is written out separately by the special sorter, some information is necessarily lost. As in Figure 18c, the two cases shown are indistinguishable if the faces are written out separately, even though the further face in the left case should not be visible at all. Surtherland has proposed a new visible surface algorithm [11] that combines the best features of the Newell and Watkins approaches; in particular, the algorithm can supply an ordered list of

segments so that transparency calculations are feasible,
and also exactly what segment is visible at each sample point,
to avoid the problem of Figure 18c.

7. CONCLUSIONS

We have described a series of algorithms that together implement a system for visible surface display. Solutions to the polygon clipping and sampling error problems have been demonstrated, which are also usable with other visible surface algorithms. The Newell special sorter has been closely examined and shown to always work. The visible surface system has been partially implemented on a PDP 8e minicomputer, as described in the Appendix.

APPENDIX

Figures 4 and 19 were generated with a preliminary version of the visible surface algorithm. Figure 20 is a description of the available hardware in the Illiac III system, part of which we used. The raster frame buffer was stored in one of the Fabritek core modules; its 128K bytes is sufficient to hold a 256 x 512 picture. The other module was used as a very high speed "disk" (750 nsec access time for 80 bits), and contained the PDP 8e operating system along with a "virtual memory" file specifying object description and manipulation data. The raster frame buffer is displayed on the monitor; the frame rate is about 1/2 second, but the slow-decay phosphor allows non-photographic observation of the rendering.

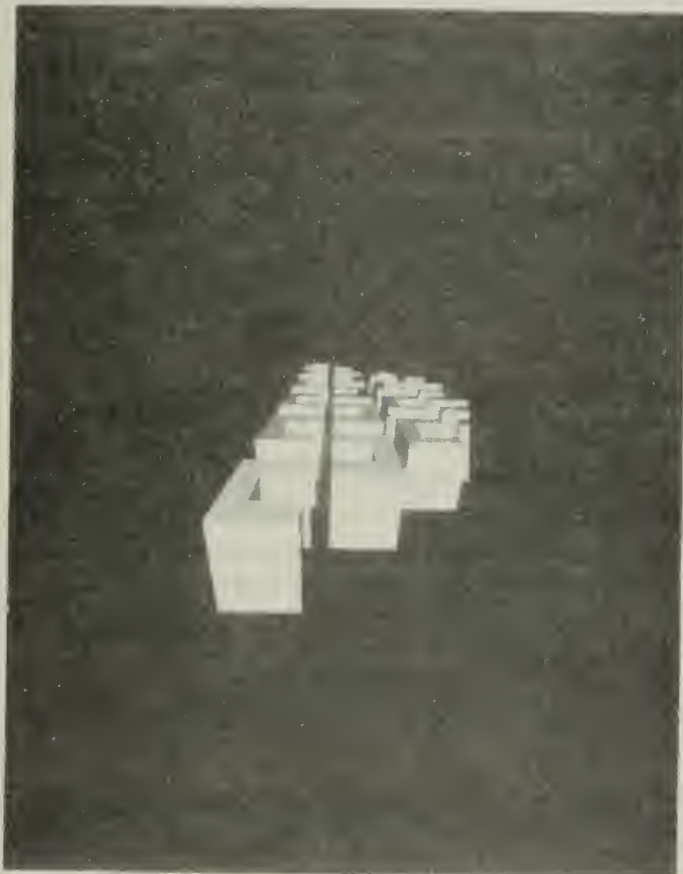
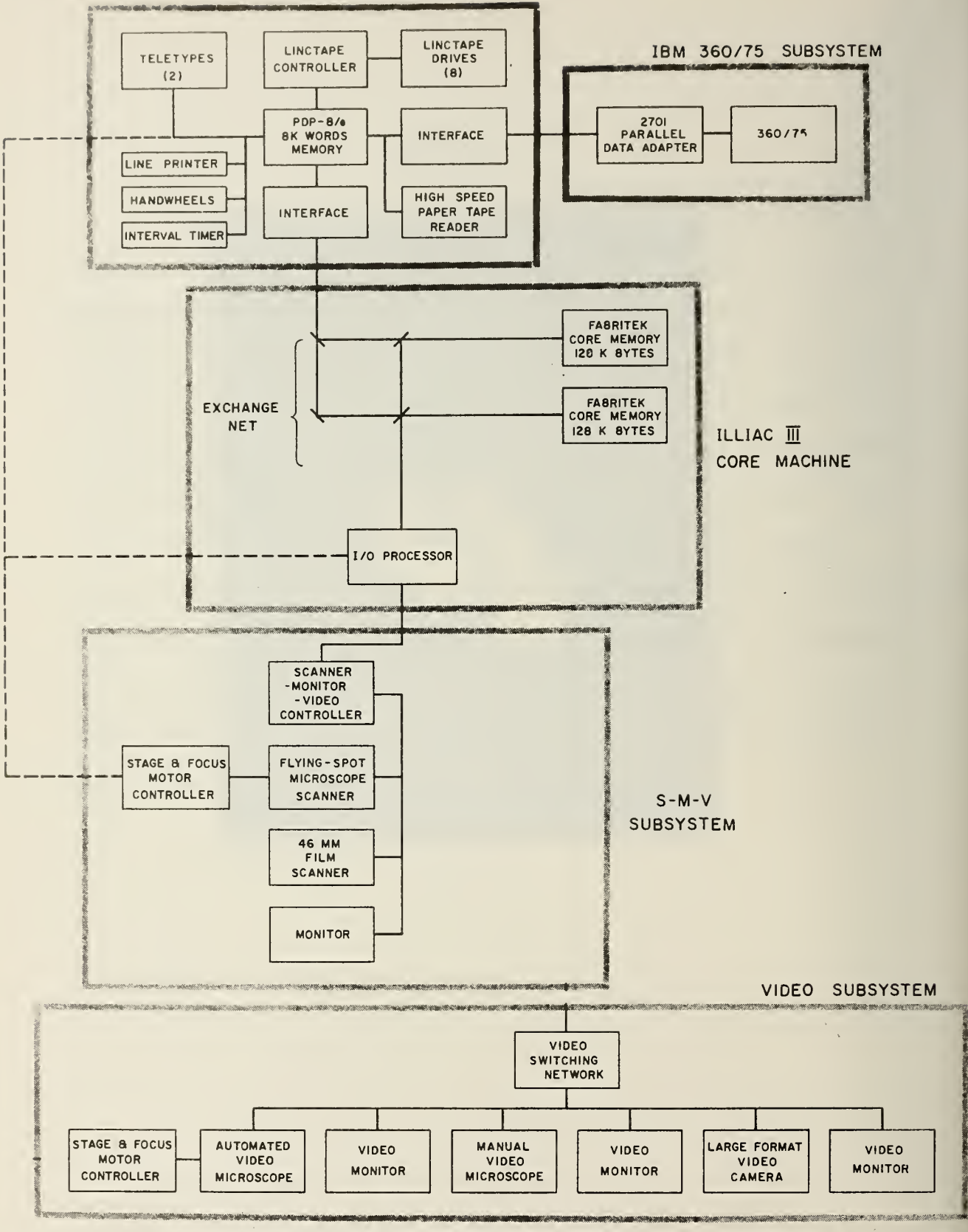


Figure 19. Figure 4 Rotated and Clipped

DEC PDP-8/e SUBSYSTEM

IBM 360/75 SUBSYSTEM



———— DATA AND/OR CONTROL
 - - - - - CONTROL ONLY

Figure 20. Operable Hardware

LIST OF REFERENCES

- [1] Newell, M. E., R. G. Newell and T. L. Sancha, "A Solution to the Hidden Surface Problem," ACM National Conference Proceedings, 1972.
- [2] Gourad, H., "Computer Display of Curved Surfaces," University of Utah, Technical Report CSC-71-113, June 1971.
- [3] Mahl, R., "Visible Surface Algorithms for Quadric Patches," University of Utah, Technical Report CSC-70-111, December 1970.
- [4] Romney, G. W., "Computer Assisted Assembly and Rendering of Solids," RADC-TR-69-365 Technical Report, September 1969.
- [5] Warnock, J. E., "A Hidden Surface Algorithm for Computer Generated Halftime Pictures," RADC-TR-69-249, June 1969.
- [6] Watkins, G. S., "A Real Time Visible Surface Algorithm," University of Utah, Technical Report, CSC-70-101, June 1970.
- [7] Bouknight, W. J., "An Improved Procedure for Generation of Three-dimensional Halftimed Computer Graphics Representations," CACM, Vol. 13, p. 527, September 1970.
- [8] Schumacker, R. A., et. al., "Study for Applying Computer-Generated Images to Visual Simulation," AFHRL-TR-69-14, U.S. Air Force Human Resources Laboratory, September 1969.
- [9] Newman, W. M. and R. F. Sproull, "Principles of Interactive Computer Graphics," McGraw-Hill Computer Science Series, 1973, pp. 252-253.
- [10] Roberts, L. G., "Picture Coding Using Pseudo-random Noise," IRE Trans. Information Theory, Vol. IT-8, February 1962, pp. 145-154.
- [11] Sutherland, I. E., R. F. Sproull and R. A. Schumacker, "Sorting and the Hidden-surface Problem," AFIPS National Computer Conference, 1973.

U. S. ATOMIC ENERGY COMMISSION
UNIVERSITY-TYPE CONTRACTOR'S RECOMMENDATION FOR
DISPOSITION OF SCIENTIFIC AND TECHNICAL DOCUMENT

(See Instructions on Reverse Side)

1. AEC REPORT NO.

COO-2118-0046

2. TITLE

DESIGN OF
AN ALGORITHM FOR THE DISPLAY
OF VISIBLE SURFACES

3. TYPE OF DOCUMENT (Check one):

- a. Scientific and technical report
 b. Conference paper not to be published in a journal:
Title of conference _____
Date of conference _____
Exact location of conference _____
Sponsoring organization _____
 c. Other (Specify) _____

4. RECOMMENDED ANNOUNCEMENT AND DISTRIBUTION (Check one):

- a. AEC's normal announcement and distribution procedures may be followed.
 b. Make available only within AEC and to AEC contractors and other U.S. Government agencies and their contractors.
 c. Make no announcement or distribution.

5. REASON FOR RECOMMENDED RESTRICTIONS:

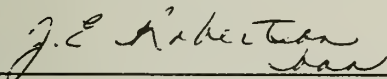
6. SUBMITTED BY: NAME AND POSITION (Please print or type)

J. E. Robertson
Acting Principal Investigator

Organization

Department of Computer Science
University of Illinois
Urbana, Illinois 61801

Signature



Date

October 1973

FOR AEC USE ONLY

7. AEC CONTRACT ADMINISTRATOR'S COMMENTS, IF ANY, ON ABOVE ANNOUNCEMENT AND DISTRIBUTION RECOMMENDATION:

8. PATENT CLEARANCE:

- a. AEC patent clearance has been granted by responsible AEC patent group.
 b. Report has been sent to responsible AEC patent group for clearance.
 c. Patent clearance not required.

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-73-592	2.	3. Recipient's Accession No.
4. Title and Subtitle DESIGN OF AN ALGORITHM FOR THE DISPLAY OF VISIBLE SURFACES		5. Report Date October 1973	
7. Author(s) Walt Donovan		6.	
9. Performing Organization Name and Address Department of Computer Science University of Illinois Urbana, Illinois 61801		8. Performing Organization Rept. No.	
12. Sponsoring Organization Name and Address US AEC Chicago Operations Office 9800 South Cass Avenue Argonne, Illinois		10. Project/Task/Work Unit No. 11. Contract/Grant No. US AEC AT(11-1)2118	
15. Supplementary Notes		13. Type of Report & Period Covered thesis	
16. Abstracts A visible surface algorithm capable of showing on a raster display smooth, transparent objects without jagged edges is described. The polygons defining the object to be displayed are put into visible order, and then output to the raster frame buffer by a special face drawing and shading algorithm. Very large objects or collections of objects can be manually put into visible order and each group drawn separately; this allows scenes with more faces than can be held in the program data structure at any one time. Minimum hardware for implementation is a raster display with a frame buffer, and a minicomputer with a disk.		14.	
17. Key Words and Document Analysis. 17a. Descriptors visible surface algorithm face sorting polygon clipping jagged edge removed smooth shading transparent object display raster displays scan line techniques 17b. Identifiers/Open-Ended Terms 17c. COSATI Field/Group			
18. Availability Statement unlimited distribution		19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages
		20. Security Class (This Page) UNCLASSIFIED	22. Price

JAN 1 1977

NOV 30 1973

1973



FEB 20 1975

UNIVERSITY OF ILLINOIS-URBANA
510.84 IL&R no. C002 no.590-594(1973
Design of totally self-checking asynchro



3 0112 088400897